Chapter 10

Predefined Macros and Built-In Functions

This chapter describes the following topics:

- Predefined macros
- Built-in functions

VAX C predefines these macros and functions for your programming convenience. The macros assist in transporting code and performing simple tasks that are common to many programs. The built-in functions access VAX instructions very efficiently.

10.1 Predefined Macros

The following sections describe the VAX C predefined macros for use in your programs.

10.1.1 System-Identification Macros

VAX C automatically defines macros that can be used to identify the system on which the program is running. These macros can assist in writing code that executes conditionally, depending on whether the program is running on a Digital system or some other system. These symbols are defined as if the following text fragment were included by the compiler before every compilation source group:

```
#define bsd4_2          1
#define ultrix          1
#define unix            1
#define vax             1
#define VAX             1
#define vaxc            1
#define VAXC            1
#define vax11c          1
#define VAX11C          1
```

You can use these definitions to separate portable and nonportable code in any of your VAX C programs.

You can use these symbols to conditionally compile VAX C programs used on more than one operating system to take advantage of system-specific features. See Section 9.2 for more information about using the preprocessor conditional compilation directives.

Consider the following example:

```
#if      VAXC
#include <descript.h>       /* Include descriptor definitions  */
#endif
```

10.1.2 CC$gfloat (G_Floating Identification Macro)

VAX C automatically defines a macro that can be used to identify whether you are compiling your program using the G_floating option. This macro can assist in writing code that executes conditionally, depending on whether the program is running using D_floating or G_floating precision.

If you compile your program using the **–Mg** option, this symbol is defined as if the following were included before every compilation source group:

```
#define  CC$gfloat  1
```

If you did not compile your program using the **–Mg** option, this symbol is defined as if the following were included before every compilation source group:

```
#define  CC$gfloat  0
```

You can conditionally assign values to variables of type **double** without causing an error and without being certain of how much storage was allocated for the variable. For example, external variables may be assigned values as follows:

```
#if CC$gfloat
double x = 0.12e308;          /* Range to 10 to the 308th power */
#else
double x = 0.12e38;           /* Range to 10 to the 38th power  */
#endif
```

The VAX C compiler determines whether or not to substitute the value 1 for every occurrence of the predefined identifiers in a program; these identifiers are reserved by Digital. The effect of these definitions may be removed by explicitly undefining the conflicting name. See Section 9.1.4 for more information about undefining. For more information about the G_floating representation of the **double** data type, see Chapter 7.

10.1.3 The __DATE__ Macro

The __DATE__ macro evaluates to a string specifying the date on which the compilation started. The string presents the date in the following format:

Mmm-dd-yyyy

The first d is a space if dd is less than 10.

The following is an example of the __DATE__ macro:

```
printf("%s",_ _DATE_ _);
```

10.1.4 The __FILE__ Macro

The __FILE__ macro evaluates to a string specifying the file specification of the current source file. The following is an example of the __FILE__ macro:

```
printf("file %s" _ _FILE_ _);
```

10.1.5 The __LINE__ Macro

The __LINE__ macro evaluates to an integer specifying the number of the line in the source file containing the macro reference. The following is an example of the __LINE__ macro:

```
printf("At line %d in file %s", _ _LINE_ _, _ _FILE_ _);
```

10.1.6 The __TIME__ Macro

The __TIME__ macro evaluates to a string specifying when the compilation started. The string presents the time in the following format:

hh:mm:ss

The following is an example of the __TIME__ macro:

```
printf("%s", _ _TIME_ _);
```

10.2 Built-In Functions

The following sections describe the built-in functions that allow you to directly access the VAX hardware and machine instructions to perform operations that are cumbersome, slow, or impossible in pure C.

These functions are very efficient because they are built into the VAX C compiler. This means that a call to one of these functions does not result in a reference to a function in the run-time library or to a function in your program. Instead, the compiler generates the machine instructions necessary to carry out the function directly at the call site. Because most of these built-in functions closely correspond to single VAX machine instructions, the result is small, fast code.

Some of these built-in functions (such as those that operate on strings or bits) are of general interest. Others (such as the functions dealing with process context) are of interest if you are writing device drivers or other privileged software. Some of the functions discussed in the following sections are privileged and unavailable to user mode programs.

You must place the following pragma in your source file before using one or more built-in functions:

```
#pragma builtins
```

Some of the built-in functions have optional arguments or allow a particular argument to have one of many different types. To describe the different legal combinations of arguments, the description of each built-in function may list several different prototypes for the function. As long as a call to a built-in function matches one of the prototypes listed, the call is legal. Furthermore, any legal call to a built-in function acts as if the corresponding prototype were in scope. Thus, the compiler performs the argument checking and argument conversions specified by that prototype.

The majority of the built-in functions are named after the VAX instruction that they generate. The built-in functions provide direct and unencumbered access to those VAX instructions. Any inherent limitations to those instructions are limitations to the built-in functions as well. For instance, the MOVC3 instruction and the _MOVC3 built-in function can move at most 65,535 characters.

10.2.1 Add Aligned Word Interlocked (_ADAWI)

The _ADAWI function adds its source operand to the destination. This function is interlocked against similar operations by other processors or devices in the system.

The _ADAWI function has the following formats:

```
int _ADAWI(short src, short *dest);
int _ADAWI(short src, unsigned short *dest);
```

src
Is the value to be added to the destination.

dest
Is a pointer to the destination. The destination must be aligned on a word boundary. (One way to achieve alignment is to use **_align**.)

There are three possible return values, as follows:

- –1, if the sum when considered to be a signed number is negative
- 0, if the sum is zero
- 1, if the sum is positive

10.2.2 Branch on Bit Clear-Clear Interlocked (_BBCCI)

The _BBCCI function performs the following functions in interlocked fashion:

- Returns the complement of the bit specified by the two arguments
- Clears the bit specified by the two arguments

The _BBCCI function has the following format:

```
int _BBCCI(int position, void *address);
```

position
Is the position of the bit within the field.

address
Is the base address of the field.

The return value is 0 or 1, which is the complement of the value of the specified bit before being cleared.

10.2.3 Branch on Bit Set-Set Interlocked (_BBSSI)

The _BBSSI function performs the following functions in interlocked fashion:

- Returns the status of the bit specified by the two arguments
- Sets the bit specified by the two arguments

The _BBSSI function has the following format:

```
int _BBSSI(int position, void *address);
```

position
Is the position of the bit within the field.

address
Is the base address of the field.

The return value is 0 or 1, which is the value of the specified bit before being set.

10.2.4 Find First Clear Bit (_FFC)

The _FFC function finds the position of the first clear bit in a field. The bits are tested for clear status starting at bit 0 and extending to the highest bit in the field.

The _FFC function has the following format:

```
int _FFC(int start, char size, const void *base, int *position);
```

start
Is the start position of the field.

size
Is the size of the field, in bits. The size must be a value from 0 to 32 bits.

base
Is the address of the field.

position
Is the address of an integer to receive the position of the clear bit. If no bit is clear, the integer is set to the position of the first bit to the left of the last bit tested.

There are two possible return values, as follows:

- 0, if all bits in the field are set
- 1, if a bit with value 0 is found

10.2.5 Find First Set Bit (_FFS)

The _FFS function finds the position of the first set bit in a field. The bits are tested for set status starting at bit 0 and extending to the highest bit in the field.

The _FFS function has the following format:

```
int _FFS(int start, char size, const void *base, int *position);
```

start
Is the start position of the field.

size
Is the size of the field, in bits. The size must be a value from 0 to 32 bits.

base
Is the address of the field.

position
Is the address of an **int** to receive the position of the set bit. If no bit is set, the integer is set to the position of the first bit to the left of the last bit tested.

There are two possible return values, as follows:

- 0, if all bits in the field are clear
- 1, if a bit with value 1 is found

10.2.6 Halt (_HALT)

The _HALT function halts the processor when executed by a process running in kernel mode. This is a privileged function.

The _HALT function has the following format:

```
void _HALT(void);
```

10.2.7 Insert Entry into Queue at Head Interlocked (_INSQHI)

The _INSQHI function inserts an entry into the front of a queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _INSQHI function has the following format:

```
int _INSQHI(void *new_entry, void *head);
```

new_entry
Is a pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

head
Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

There are three possible return values, as follows:

- 0, if the entry was inserted, but it was not the only entry in the list
- 1, if the entry was not inserted because the secondary interlock failed
- 2, if the entry was inserted and it was the only entry in the list

10.2.8 Insert Entry into Queue at Tail Interlocked (_INSQTI)

The _INSQTI function inserts an entry at the end of a queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _INSQTI function has the following format:

```
int _INSQTI(void *new_entry, void *head);
```

new_entry
Is a pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

head
Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

There are three possible return values, as follows:

- 0, if the entry was inserted, but it was not the only entry in the list
- 1, if the entry was not inserted because the secondary interlock failed
- 2, if the entry was inserted and it was the only entry in the list

10.2.9 Insert Entry in Queue (_INSQUE)

The _INSQUE function inserts a new entry into a queue following an existing entry.

The _INSQUE function has the following format:

```
int _INSQUE(void *new_entry, void *predecessor);
```

new_entry
Is a pointer to the new entry to be inserted.

predecessor
Is a pointer to an existing entry in the queue.

There are two possible return values, as follows:

- 0, if the entry was the only entry in the queue
- 1, if the entry was not the only entry in the queue

10.2.10 Load Process Context (_LDPCTX)

The _LDPCTX function restores the register and memory-management context. This is a privileged function.

The _LDPCTX function has the following format:

```
void _LDPCTX(void);
```

10.2.11 Locate Character (_LOCC)

The _LOCC function locates the first character in a string matching the target character.

The _LOCC function has the following formats:

```
int _LOCC(char target, unsigned short length,
          const char *string);
```

```
int _LOCC(char target, unsigned short length,
          const char *string, char **position);
```

target
Is the character being searched.

length
Is the length of the searched string. The length must be a value from 0 to 65,535.

string
Is a pointer to the searched string.

position
Is a pointer to a pointer to a character. If the searched character is found, the pointer pointed to by position is updated to point to the character found. If the character is not found, the pointer pointed to by position is set to the address one byte beyond the string. This is an optional argument.

If the target character is found, the return value is the number of bytes remaining in the string; otherwise, the return value is 0.

10.2.12 Move from Processor Register (_MFPR)

The _MFPR function returns the contents of a processor register. This is a privileged function.

The _MFPR function has the following formats:

```
void _MFPR(int register_num, int *destination);
void _MFPR(int register_num, unsigned int *destination);
```

register_num
Is the number of the privileged register to be read.

destination
Is a pointer to the location receiving the value from the register. This location may be a **signed** or **unsigned int**.

10.2.13 Move Character 3 Operand (_MOVC3)

The _MOVC3 function copies a block of memory. It is the preferred way to copy a block of memory to a new location.

The _MOVC3 function has the following formats:

```
void _MOVC3(unsigned short length, const char *src, char *dest);

void _MOVC3(unsigned short length, const char *src, char *dest,
            char **endsrc);

void _MOVC3(unsigned short length, const char *src, char *dest,
            char **endsrc, char **enddest);
```

length
Is the length of the source string, in bytes. The length must be a value from 0 to 65,535.

src
Is a pointer to the source string.

dest
Is a pointer to the destination memory.

endsrc
Is a pointer to a pointer. The _MOVC3 function sets the pointer that is pointed to by endsrc pointing to the address of the byte beyond the source string. It is optional if the enddest argument is not given.

enddest
Is a pointer to a pointer. The _MOVC3 function sets the pointer pointed to by endsrc to the address of the byte beyond the destination string. This is an optional argument.

10.2.14 Move Character 5 Operand (_MOVC5)

The _MOVC5 function allows the source string specified by the pointer and length pair to be moved to the destination string specified by the other pointer and length pair. If the source string is smaller than the destination string, the destination string is padded with the specified character.

The _MOVC5 function has the following formats:

```
void _MOVC5(unsigned short srclen, const char *src, char fill,
               unsigned short destlen, char *dest);

void _MOVC5(unsigned short srclen, const char *src, char fill,
               unsigned short destlen, char *dest,
      unsigned short *unmoved_src);

void _MOVC5(unsigned short srclen, const char *src, char fill,
               unsigned short destlen, char *dest,
      unsigned short *unmoved_src, char **endsrc);

void _MOVC5(unsigned short srclen, const char *src, char fill,
               unsigned short destlen, char *dest,
      unsigned short *unmoved_src, char **endsrc,
               char **enddest);
```

srclen
Is the length of the source string, in bytes. The length must be a value from 0 to 65,535.

src
Is a pointer to the source string.

fill
Is the fill character to be used if the source string is smaller than the destination string.

destlen
Is the length of the destination string, in bytes. The length must be a value from 0 to 65,535.

dest
Is a pointer to the destination string.

unmoved_src
Is a pointer to a short integer that the _MOVC5 function sets to the number of unmoved bytes remaining in the source string.

endsrc
Is a pointer to a pointer. The _MOVC5 function sets the pointer pointed to by endsrc pointing to the address of the byte beyond the source string. It is optional if the enddest argument is not given.

enddest
Is a pointer to a pointer. The _MOVC5 function sets the pointer pointed to by endsrc to the address of the byte beyond the destination string. This is an optional argument.

10.2.15 Move from Processor Status Longword (_MOVPSL)

The _MOVPSL function stores the value of the Processor Status Longword (PSL).

The _MOVPSL function has the following formats:

```
void _MOVPSL(int *psl);
void _MOVPSL(unsigned int *psl);
```

psl
Is the address of the location for storing the value of the Processor Status Longword.

10.2.16 Move to Processor Register (_MTPR)

The _MTPR function loads a value into one of the special processor registers. It is a privileged function.

The _MTPR function has the following format:

```
int _MTPR(int src, int register_num);
```

src
Is the value to store into the processor register.

register_num
Is the number of a privileged register to be updated.

The return value is the V condition flag from the Processor Status Longword (PSL).

10.2.17 Probe Read Accessibility (_PROBER)

The _PROBER function checks to see if you can read the first and last byte of the given address and length pair.

The _PROBER function has the following format:

```
int _PROBER(char mode, unsigned short length, const void *address);
```

mode
Is the processor mode used for checking the access.

length
Is the length of the memory segment, in bytes. The length must be a value from 0 to 65,535.

address
Is the pointer to the memory segment to be tested for read access.

There are two possible return values, as follows:

- 0, if both bytes are not accessible
- 1, if both bytes are accessible

10.2.18 Probe Write Accessibility (_PROBEW)

The _PROBEW function checks the write accessibility of the first and last byte of the given address and length pair.

The _PROBEW function has the following format:

```
int _PROBEW(char mode, unsigned short length, const void *address);
```

mode
Is the processor mode used for checking the access.

length
Is the length of the memory segment, in bytes. The length must be a value from 0 to 65,535.

address
Is the pointer to the memory segment to be tested for write access.

There are two possible return values, as follows:

- 0, if both bytes are not accessible
- 1, if both bytes are accessible

10.2.19 Read General-Purpose Register (_READ_GPR)

The _READ_GPR function returns the value of a general-purpose register.

The _READ_GPR function has the following format:

```
int _READ_GPR(int register_num);
```

register_num
Is an integer constant expression giving the number of the general-purpose register to be read.

The return value is the value of the general-purpose register.

10.2.20 Remove Entry from Queue at Head Interlocked (_REMQHI)

The _REMQHI function removes the first entry from the queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _REMQHI function has the following format:

```
int _REMQHI(void *head, void **removed_entry);
```

head
Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

removed_entry
Is a pointer to a pointer that _REMQHI sets to point to the removed entry.

There are four possible return values, as follows:

- 0, if the entry was removed and the queue has remaining entries
- 1, if the entry could not be removed because the secondary interlock failed
- 2, if the entry was removed and the queue is now empty
- 3, if the queue was empty

10.2.21 Remove Entry from Queue at Tail Interlocked (_REMQTI)

The _REMQTI function removes the last entry from the queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _REMQTI function has the following format:

```
int _REMQTI(void *head, void **removed_entry);
```

head
Is a pointer to the queue header. The header must be aligned on a quadword boundary. (One way to achieve alignment is to use **_align**.)

removed_entry
Is a pointer to a pointer that _REMQTI sets to point to the removed entry.

There are four possible return values, as follows:

- 0, if the entry was removed and the queue has remaining entries
- 1, if the entry could not be removed because the secondary interlock failed
- 2, if the entry was removed and the queue is now empty
- 3, if the queue was empty

10.2.22 Remove Entry from Queue (_REMQUE)

The _REMQUE function removes an entry from a queue.

The _REMQUE function has the following format:

```
int _REMQUE(void *entry, void **removed_entry);
```

entry
Is a pointer to the queue entry to be removed.

removed_entry
Is a pointer to a pointer that _REMQUE sets to the address of the entry removed from the queue.

There are three possible return values, as follows:

- 0, if the entry was removed and the queue has remaining entries

- 1, if the entry was removed and the queue is now empty
- 2, if the queue was empty

10.2.23 Scan Characters (_SCANC)

The _SCANC function locates the first character in a string with the desired attributes. The attributes are specified through a table and a mask.

The _SCANC function has the following formats:

```
int _SCANC(unsigned short length, const char *string,
           const char *table, char mask);

int _SCANC(unsigned short length, const char *string,
           const char *table, char mask, char **match);
```

length
Is the length of the string to scan, in bytes. The length must be a value from 0 to 65,535.

string
Is a pointer to the string to scan.

table
Is a pointer to the table.

mask
Is the mask.

match
Is a pointer to a pointer that the _SCANC function sets to the address of the byte that matched. (If no match occurs, it is set to the address of the byte following the string.) This is an optional argument.

The return value is the number of bytes remaining in the string if a match was found; otherwise, the return value is 0.

10.2.24 Simple Read (_SIMPLE_READ)

The _SIMPLE_READ function reads I/O registers or shared memory. It causes a MOVB, MOVW, or MOVL instruction to be generated that cannot be moved or modified during optimization.

The _SIMPLE_READ function has the following formats:

```
char _SIMPLE_READ(const char *source);
short _SIMPLE_READ(const short *source);
int _SIMPLE_READ(const int *source);
long _SIMPLE_READ(const long *source);
```

source
Is a pointer to the source to be read. The object being pointed to must be a signed integer. The type of the object pointed to determines the type of the function result.

The return value is the value of the specified source.

10.2.25 Simple Write (_SIMPLE_WRITE)

The _SIMPLE_WRITE function writes to I/O registers or shared memory. It causes a MOVB, MOVW, or MOVL instruction to be generated that cannot be moved or modified during optimization.

The _SIMPLE_WRITE function has the following formats:

```
void _SIMPLE_WRITE(char value, char *dest);
void _SIMPLE_WRITE(short value, short *dest);
void _SIMPLE_WRITE(int value, int *dest);
void _SIMPLE_WRITE(long value, long *dest);
```

value
Is the value to be stored. The type of the destination argument determines the type of this argument.

dest
Is a pointer to the destination. The type of the object pointed to by dest must be a signed integer type. The type of this object determines the type of the first argument to this function.

10.2.26 Skip Character (_SKPC)

The _SKPC function locates the first character in a string that does not match the target character.

The _SKPC function has the following formats:

```
int _SKPC(char target, unsigned short length, const char *string);

int _SKPC(char target, unsigned short length, const char *string,
          char **position);
```

target
Is the target character.

length
Is the length of the string, in bytes. The length must be a value from 0 to 65,535.

string
Is a pointer to the string to scan.

position
Is a pointer to a pointer. The _SKPC function sets the pointer pointed to by position to the address of the nonmatching character. (If all the characters match, it is set to the address of the first byte beyond the string.) This is an optional argument.

The return value is the number of bytes remaining in the string if an unequal byte was located; otherwise, the return value is 0.

10.2.27 Span Characters (_SPANC)

The _SPANC function locates the first character in a string without certain attributes. The attributes are specified through a table and a mask.

The _SPANC function has the following formats:

```
int _SPANC(unsigned short length, const char *string,
           const char *table, char mask);
```

```
int _SPANC(unsigned short length, const char *string,
           const char *table, char mask, char **position);
```

length
Is the length of the string, in bytes. The length must be a value from 0 to 65,535.

string
Is a pointer. It points to the string to be scanned.

table
Is a pointer to the table.

mask
Is the mask.

position
Is a pointer to a pointer. The _SPANC function sets the pointer pointed to by position to the address of the byte that does not match the attributes. (If all the characters in the string match, this pointer is set to the address of the first byte beyond the string.) This is an optional argument.

The return value is the number of bytes remaining in the string if a match was found; otherwise, the return value is 0.

10.2.28 Save Process Context (_SVPCTX)

The _SVPCTX function saves the context of a process. The general-purpose registers are saved in the process control block, which is later used to resume a process. This function is privileged.

The _SVPCTX function has the following format:

```
void _SVPCTX(void);
```

10.2.29 Write General-Purpose Register (_WRITE_GPR)

The _WRITE_GPR function loads a value into a specified general-purpose register.

The _WRITE_GPR function has the following format:

```
void _WRITE_GPR(int value, int register_num);
```

value
Is the value to load into the register.

register_num
Is an integer constant expression giving the number of the general-purpose register to be loaded. The register number must be a value from 0 to 15.